Karlsruhe Institute of Technology

Building and optimizing a small cluster for the Student Cluster Competition

Thesis of

Jos Ewert

At the Faculty of Mathematics
Institute for Applied and
Numerical Mathematics 4

Reviewer:	Prof. Dr. Vincent Heuveline
Second reviewer:	Prof. Dr. Wolfgang Karl

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, den 15. Oktober 2012

Unterschrift: .....

Contents

1 Introduction

The Student Cluster Competition or Student Cluster Challenge (SCC) is a biannual event organized by the HPC Advisory Council where international teams of 6 undergraduate students build and manage their own small cluster. The clusters are presented at the Supercomputing Conference (SC) in the USA or at the International Super Computing Conference (ISC) in Germany. During these conferences the clusters are benchmarked with HPCC, a collection of benchmarks including LINPACK, which is also used to rate the TOP500 supercomputers [1], and several applications "to demonstrate the incredible capabilities of state-of-the-art high-performance cluster hardware and software" [2]. The competition was held for the first time at the ISC in 2012, where the competing teams were the University of Colorado and Stony Brook University from the USA, Tsinghua University and NUDT from China, and the Karlsruhe Institute of Technology (KIT) from Germany.

The Engineering Mathematics and Computing Lab (EMCL) at the KIT took part in the competition for the first time, and during the approximately 9 months of preparation for the SCC, their team changed a few times. I joined the team several months before the competition and the members of the team for the last few months, that eventually went to the competition on the ISC'12, were Markus Grotz, Pascal Kraft, Robin Lutz, Claudia Pfreundt, Christian Winkler, and myself Jos Ewert.

I was mostly concerned with the cluster's hardware, installing the operating system, optimizing OpenFOAM, which was one of the benchmark applications, and I also configured the HPCC benchmark.

In this paper, I will describe what the team and I tested and planned to have the best cluster for the competition. Section 2 starts with the rules of the competition and ends with the conclusions we took from these rules. In Section 3, I present the choices we had for our hardware and the optimizations I tried on our final system's hardware. Section 4 covers our software stack to manage and run the cluster, Section 5 contains a short presentation of our HPC Challenge Benchmark (HPCC) results and finally OpenFOAM is presented in Section 6.

2 Rules of the Student Cluster Competition

For the competition we had to follow certain rules which could be reviewed on the competition's website [3] and were later on clarified but also changed in the FAQ [4].

Any team of 6 undergraduate students could compete with a cluster that was limited to a maximum of 3 kW power usage, but was otherwise not restricted on the software or hardware. We were free to use experimental or unreleased software and hardware. During the 3 days at the ISC'12 no one was allowed to touch or power down the systems. Several benchmarks were executed to determine the winners of the "Overall Winner" and the "Highest LINPACK" trophy, in addition a "Fan Favorite" trophy was awarded to the team with the highest number of votes from the ISC's audience. The benchmarks were the HPC Challenge benchmark (HPCC), which includes LINPACK (HPL), four known applications that were OpenFOAM, CPMD, CP2K, and Nemo, and also two unknown applications that were revealed during the event at the ISC. HPCC defined the hardware baseline for the rest of the benchmarks, making only benchmarks valid that were executed on the exact same hardware. HPCC also produced the LINPACK score, which is the number of floating point operations per second the system can perform, and counted for 10% of the overall score, the performance of the six applications added 80% to the score, and the last 10% were interviews taken during the competition. The applications' performance was at first to be gauged by counting the amount of jobs a cluster could complete over the

course of the day. However, this was changed only a few weeks before the competition to measure the runtime of a few selected jobs that could, if necessary, be executed several times with different configurations to achieve the best time possible. In addition HPL could be executed on its own to achieve a higher LINPACK score, that was, however, only used for the "Highest LINPACK" trophy.

The rules stated that the power usage was measured with an APC AP7821 PDU (power distribution unit) and also enforced with a circuit breaker limit of 3 kW; no power source other than the provided circuit was allowed for the cluster. Each team additionally had a separate circuit for systems and equipment that were not actively involved in the execution of benchmarks, like screens or monitoring systems. Different to the rules, during the competition at the ISC we were informed that we could have power usage peaks between 3 kW and 3.1 kW that could last for up to one minute and that the circuit breaker would limit at 3.5 kW.

Teams could apply for the competition until Friday, October 21, 2011, and if they were accepted into the competition, had to submit the final architecture of the cluster before Monday, April 30, 2012. The competition itself was from Monday, June 18, 2012 to Wednesday, June 20, 2012.

After evaluating the rules, we tried to find a design that was capable of executing the highest number of jobs and, if necessary, execute those jobs in parallel that had datasets that were too small to utilize the cluster efficiently. This meant that we did not try to achieve the best possible run time of a job, but the highest possible efficiency in order to have a greater number of nodes in our cluster. As we were not allowed to touch or shut down parts of the cluster, we could not build specialized nodes for each application, thus we had to prioritize which application we would optimize the power usage to. After some testing we decided that OpenFOAM would be our target application as it had significantly less power usage than LINPACK, but only slightly more power usage than the other applications. This is explained more in depth in Subsection 3.2. We assumed that we could not exceed the 3 kW limit without risking the circuit breakers shutting down the system, so we always considered the highest peak power usage while evaluating the applications and the system.

3 Hardware

We had nearly free reign over what kind of hardware we would use, so we spent much time trying to find the best hardware setup to suit the competition's rules and applications. The following presents the first systems we considered in Subsection 3.1 and the various setups for the final system I tested in Subsection 3.2.

3.1 Initial Systems

During the preparation for the competition we considered several systems, including turnkey solutions like a large shared memory cluster from Numascale [5] or a PowerPC (PPC) cluster from IBM. We also considered more custom setups with low power CPUs like an ARM based cluster or an Intel Atom based cluster, as opposed to regular server CPUs like the Intel Xeon. We looked at hybrid approaches for our custom setups combining x86 processors and GPUs from either Nvidia or AMD. We evaluated interconnects like 10-gigabit Ethernet and InfiniBand. Finally we considered ways to power down components depending on the currently needed resources.

While we evaluated the IBM PowerPC solutions, a Blue Gene/P, we quickly realized that their software stack, with a custom proprietary kernel [6] that only supported a subset of POSIX (for example no fork() [7]) and their XL compiler, differed quite substantially from x86 clusters running a Linux operating system (OS) with the Intel or GNU compilers.

Neither our team nor our valuable assistance from the Steinbuch Centre for Computing, who host x86 based clusters powered by a Linux OS, had any experience with this toolchain yet. Additionally, most of the open source software we needed to run for the competition was only tested with GNU or Intel compilers on Linux. We realized that we would have to learn the particularities of the system and quite possibly spend valuable time, that we could otherwise use to optimize the system, debugging and modifying the source code to adapt the software to the new toolchain.

Researching the hardware side, we realized that the IBM systems were quite capable, with the Roadrunner system still in 10th place of the TOP500 [8], but their power efficiency of around 0.5 GFlops/W was not convincing compared to other systems that could reach over 1 GFlops/W.

We also looked into a Numascale system, which is a cluster connected with a 3d or 2d torus network that is completely transparent to the operating system, and thus behaves like a large non-uniform memory access (NUMA) system. This makes it possible that software does not need MPI or other systems to communicate between the nodes and applications scale up by just creating more threads like they would on a single system. In our case however, that was no benefit, as all the required software already used MPI to communicate between nodes.

Researching the hardware, we found out that the network adapters were connected to the AMD HyperTransport. This limited that technology to AMD processors which, as our research showed, often had a higher thermal design power (TDP) [9] than Intel processors and did not support the AVX feature which greatly improves the floating point operations rate.

After having rejected the turnkey solutions rather early on, we looked at more custom configurations, and in early stages of the design we had several proposals to power down idle pieces of hardware. We could use controller chips with each CPU that could measure the power usage of the processor or even power it down, giving us a detailed view of which parts of the system were active and which parts could be shut down. For example we could turn off GPUs on applications that could not utilize the GPU. However, we were uncertain on how the hardware and software would react to components being powered up and down during runtime, as most of our hardware was not designed for such a task. In addition to the hardware problems, the sheer complexity of this setup and the rules of the competition, which meant no powering down the system, made such an effort impossible.

We evaluated interconnects and found studies from the HPC Advisory Council [10, 11] that show that InfiniBand had advantages even at smaller scale clusters and allowed for a faster execution of jobs. We realized quickly that a specialized low latency solution like InfiniBand was preferable over Ethernet.

While looking at low power CPUs, like the ARM and Atom processors, we realized that they required a large number of nodes to reach the 3 kW limit, which needed a large (InfiniBand) fabric to communicate that might even use more power than the processing cores. For example if we took an Intel Atom N2600 processor, we would likely have had less than 15 W of power usage per node, 3.5 W for the processor, several watt for the InfiniBand card, and a few watt for the rest of the electronics. Such a low power usage would mean around 200 nodes could be supported, however a switching system that size needed around 2300 W [12]. In addition to the very high percentage of power that the interconnect would need, low power CPU's small memory bandwidth, compared to server CPUs, would slow down the interconnect and processing in general. Moreover it seemed that there were no, or only few, ARM processors that supported the needed PCI Express bandwidth for the InfiniBand cards. We came to the conclusion that we would have had an

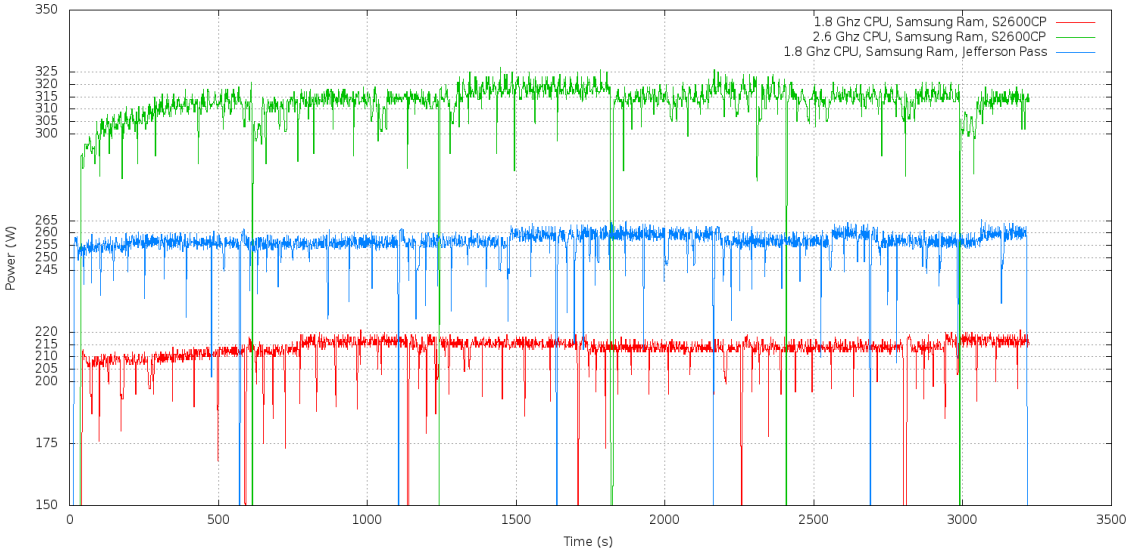


Figure 3.1: The power usage of the S2600CP and Jefferson Pass system

underperforming fabric that used more power than the actual processors, thus nullifying any kind of efficiency gains we could have had by using these processors.

We also considered hybrid approaches combining CPUs and GPUs, and we noticed that software support for CUDA from Nvidia was more mature and widespread than OpenCL used by AMD. We could gain Nvidia as a sponsor for several Tesla M2090 cards to test on our system and applications. As we evaluated the applications, we realized that only LINPACK would be able to efficiently use GPUs without extensive modification of the source code, which was beyond our teams capabilities as we lacked manpower and time. While there were several implementations for support of GPUs for our applications, most of them were in early stages and had rather large drawbacks. For example some only supported a single GPU, which could be an issue if the data that needed to be processed did not fit into the memory of one card. We also saw an issue that we could likely not use our GPUs on the unknown applications unless they had the support already build in; we did not believe that we could modify these applications fast enough to produce benchmark results. You can find a more thorough analysis of OpenFOAM on the GPU in Subsection 6.1.

Apart from the software issues, we had problems getting our systems into a stable state when we used the GPUs. For example the BIOS crashed, the InfiniBand network randomly failed or the operating system had issues with a GPU installed in the system. The issues would only cease when we removed the card. The support staff at Intel speculated that the firmware of the motherboards and drivers for their new PCI Express controller integrated in the CPU were not completely tested yet and that we triggered a bug. Lastly we measured that the power draw of the GPU solutions was quite high, we measured somewhere around 250 W per card, which was more than an entire node with two CPUs. This meant that the GPUs needed to provide a rather significant speedup to justify their power usage.

With all these issues and unconvincing performance predictions from vendors, we decided against the use of GPUs.

3.2 The Intel Xeon System

After having evaluated various systems, we decided to use a setup with 2 Sandy Bridge Intel Xeon CPUs on a regular server motherboard, FDR InfiniBand from Mellanox, and

RAM, clock speed (GHz), motherboard	power (W)	avg. runtime (s)	work (Wh)
Kingston, 1.8, S2600CP	236	596	39.1
Kingston, 2.6, S2600CP	326	592	53.6
Samsung, 1.8, S2600CP	221	550	33.7
Samsung, 2.6, S2600CP	385	487	52.1
Samsung, 1.8, Jefferson Pass	265	528	38.8
Samsung, 1.8, X9DRi-F	234	513	33.3

Table 1: Work of OpenFOAM on the test systems

DDR3 1600 memory from Samsung. The memory could be used at 1.35 V as opposed to the regular 1.5 V, which we believed would gain us an increase in power efficiency.

We had 2 different models of CPUs at our disposition, which were the Intel Xeon E5-2650L clocked at 1.8 GHz with a TDP of 70 W and the E5-2670 clocked at 2.6 GHz with a TDP of 115 W. We also tested two different kinds of RAM; DDR3 1333 from Kingston running at 1.5 V and the Samsung green memory DDR3 1600 with a lower voltage of 1.35 V. Initially we had a S2600CP dual socket server motherboard from Intel, later we tested with a Jefferson Pass system having a dual socket S2600JF motherboard, and finally with a X9DRi-F dual socket motherboard from Supermicro.

We gathered the power usage data with a power measuring device and only considered the highest peak power usage, as we wanted to avoid tripping the circuit breaker at 3kW. In Figure 3.1 you can see the measurements we took on several systems by running OpenFOAM several times in a row.

I tried gauge the efficiency of the CPUs with our applications and thus determine which CPU model would be the most beneficial. As shown in Figure 3.1 , using the same motherboard, the 2.6 GHz model peaked at about 325 W and the 1.8 GHz model at about 220 W. As power usage alone does not indicate the systems efficiency, I decided to use GFlops/W for LINPACK and Wh for the other software packages to gauge their efficiency.

In early tests we noticed that OpenFOAM and LINPACK were the most power consuming applications. Initially LINPACK needed around 10% more power than OpenFOAM, which translated into 300 W on our entire cluster, while the difference between OpenFOAM and the other applications was around 5%, around 150 W on the entire system. Early tests also showed that our final nodes would each need around 230 - 380 W and we were unsure how well we could tune the power usage by clocking down the CPUs or how that would affect the applications' performance. We decided to use OpenFOAM instead of LINPACK as the benchmark for power usage because we could potentially gain an additional node that ran at the maximum clock, whereas using another benchmark instead of OpenFOAM would likely not. We decided to clock down the CPUs during a LINPACK run, so we could have the maximum clock with OpenFOAM and the other applications. This arrangement would give us the necessary safety towards the power limit, without sacrificing too much performance with the other applications.

I used the damBreak-fine tutorial job of OpenFOAM, and increased its resolution to have around 100000 cells per CPU core, which, according to our consultant, would utilize the CPU cores efficiently and simulate typical usage. I changed the amount of time simulated during the job to have a runtime of around ten minutes and repeated each run at least three times.

During the tests on the S2600CP motherboard, I had issues setting the Samsung RAM to the correct frequency and voltage and noticed strange variations in the test results after a reboot. For example in Table 1 it is recorded that with Kingston RAM and the 2.6 GHz

CPU the power usage was 326 W, but with the same setup after a reboot the test needed 375 W and only took 546 seconds leading to a work of 56.9 Wh. The power usage increased by around 50 W and the runtime was decreased by around 50 s just by rebooting. We had no way of reading the sensor data provided by IPMI because the tools did not support the model of motherboard yet, so we could never verify what the actual voltage and memory clock were. There was no official support for DDR3 1600 with a voltage of 1.35 V in this motherboard [13], so I assume that the motherboard had conflicting data, with the RAM requesting 1.35 V but the spec requesting 1.5 V and would change the frequency and maybe voltage of the RAM randomly and thus change the results. As OpenFOAM's performance is bound to the memory bandwidth [14], these effects were quite significant. This issue did not appear with the X9DRi-F motherboard from Supermicro, which could be forced to use the values for clock and voltage requested by the DIMM and we could verify the RAM voltage with IPMI and the clock using the BIOS. In Table 1 I only recorded the most efficient tests.

We can see in Table 1 using the 1.8 GHz CPU instead of the 2.6 GHz CPU in conjunction with the Kingston RAM results in a 25% increase in efficiency and about 40% with the Samsung RAM. When clocking down the 2.6 GHz CPUs to 1.8 GHz, the runtime of the application was the same as on the 1.8 GHz CPU, but there was an increase of about 10% in power usage compared to the 1.8 GHz CPU.

When comparing the two tests with the Kingston RAM, we can see that there is barely a difference between the runtimes, however the same tests with the Samsung RAM show a significant change in runtime. This is likely due to the random change in clock, as I also had a test with the Kingston RAM that needed 546 s to finish, and thus showing a similar behavior to the tests with he Samsung RAM. This made it hard to compare results and I decided to use the time that mirrors the behavior of in the system with the Samsung RAM, that is 546 s, 375 W leading to a work of 56.9 Wh for Kingston RAM with the 2.6 GHz CPU. With that result instead of the one in the table, the runtime is decreased roughly by 10% when the Samsung RAM instead of the Kingston RAM is used, increasing efficiency by 14% on the 1.8 GHz CPU and 9% with the 2.6 GHz CPU. Comparing Samsung 8GB DIMMs and 16 GB DIMMs showed no change in power usage or performance.

After having issues with the S2600CP motherboard, we heard that there might be a developer motherboard, that would allow us to force certain voltages and clocks, build into a preview Jefferson Pass system. It is an integrated 2U four node server system with a redundant 1200 W power supply, but we could only use one of the four nodes as we lacked parts. Unfortunately the BIOS did not allow us to set the voltages and we experiences the same unpredictable behavior of the memory clock, again I compared the numbers to a run not recorded in Table 1, but that had a similar runtime using the same RAM and CPU on the S2600CP board with 524 s at 236 W, 34.5 Wh. The performance was about the same, as expected due to the very similar motherboard and other hardware, however the power usage rose by about 10%, which is likely due to the system not being used efficiently with only one node.

With this Jefferson Pass system not solving our problems, I ran the remaining tests on the S2600CP motherboard again. With 1 Samsung DIMM on each CPU, the runtime of the OpenFOAM test case was around 1500 s, at 161 W, 67.1 Wh. Using 2 DIMMs per CPU, the runtime dropped to 840 s, 175 W and needed 40.8 Wh. Finally, with 4 DIMMs per CPU, the job was done in about 540 s peaked at 210 W and thus used 31.5 Wh. Later, after the system had been build, another test was done with 8 DIMMs per CPU, on that system the baseline with 4 DIMMs was a runtime 511 s and increased to 525 s with 8 DIMMs. We more than doubled the efficiency by using 4 DIMMs. However a slight increase in runtime is seen when more memory modules are used than channels. The CPU

supports four memory channels to access up to four memory modules at the same time [15]. Because each DIMM, if plugged into the correct socket, takes one channel, it follows that four DIMMs use all four channels and lead to the best performance. The decrease in performance with more DIMMs than channels is due to a higher congestion on each channel, which can lead to more memory accesses blocking each other, thus decreasing the memory's performance.

Thanks to the greater performance and energy efficiency of the Samsung RAM we decided to use four 16 GB Samsung DDR 1600 DIMMs for each CPU.

We had problems with the unpredictable behavior of the memory clock on both Intel systems, and we were not able to test the the Jefferson Pass system under load, so we chose to try the X9DRi-F motherboard from Supermicro. We managed to have the system running reliably with the Samsung RAM running as DDR3 1600 at 1.35 V and we had the most efficient test result with this setup. Our final setup was as follows, with each node having:

- a Supermicro X9DRi-F motherboard
- 2 Intel Xeon E5-2650L CPUs at 1.8 GHz
- 8 Samsung Green Memory DDR3 1600 16 GB Registered DIMMs, forming a total of 128 GB
- 1 Samsung Green Memory SSD with 256 GB storage
- 1 Mellanox ConnectX 3 VPI FDR InfiniBand card
- 1 Enermax Platimax 500 W power supply

We received the Supermicro boards after the deadline for the submission of the final architecture, so we based the decision on how many nodes we needed on the tests from the S2600CP system with the 1.8 GHz CPU. We calculated that we wanted around 5% safety towards the power limit, we hoped to gain another 5% in power usage by using a better power supply and another 5% by using the Samsung RAM at 1.35V. In addition we reserved around 200W for the peripherals, like the switches and the head-node. By calculating $(3000W - P_{peripherals})/(P_{OpenFOAM} * 1.05 * 0.95 * 0.95) = (3000W - 200W)/(221W * 1.05 * 0.95 * 0.95)$, we determined that our power limit could support 13 nodes, a Mellanox SX6025 36-port 56Gb/s InfiniBand Switch, an Ethernet switch, and one head-node used to control the cluster. The head-node had one Intel Core i7 processor, 16 GB of RAM and a 600 GB SSD. In total we had 208 cores, 1.6 TB of RAM, and 3.3 TB Storage for computing.

Our final system had enough buffer in power to accommodate another node, unfortunately it was too late to have another node and we believed that we could no longer change the architecture of our cluster. We assume that this was because the final power supplies were more power efficient than we anticipated, compared to the one we had to use during our tests.

To optimize the hardware I briefly tested settings on channel interleaving and rank interleaving. Channel interleaving distributes accesses to subsequent memory pages over the CPU's memory channels and rank interleaving distributes them over the DIMMs ranks, which are a set of DRAM, that save the data, attached to their chip select, that determines where the data is saved. This means that accessing a sequence of memory pages will likely result in being distributed over all channels and ranks, thus hiding latency as one module can already serve data while other modules are still processing their request. I started a test with both set to 1, but canceled the test after 900 s because the baseline with rank interleaving set to 8 and channel interleaving to 4 was 494 s; it seemed useless to continue

at this point. I did not test how the performance changes if rank interleaving was set to 2, which was the number of ranks on our DIMMs.

Our 2 socket system was a NUMA system [15], that means that both CPUs had their own local memory, and could only access the other CPU's memory with a performance penalty. Since accessing the local memory is faster, it is important that programs are NUMA aware, which means that they will try to allocate their data to the local node. Usually the Linux kernel allocates memory for newly created processes on the correct node, however in case multiple threads are used, the application has to tell the kernel where to allocate memory by linking to libnuma. Our applications used MPI, which parallelizes applications by creating new processes, and libnuma, so we could safely assume that their memory was allocated to the correct node. This was confirmed by the low number of NUMA misses I observed (around 20 a day). Thus I chose to set the motherboard to be NUMA as opposed to interleaved memory access or Sufficiently uniform memory access (SUMA). However, I did not test if there was a difference in performance.

4 Software

While we were still deliberating our hardware setup we had to choose what software stack would power our cluster. At the bare minimum we needed an operating system that supported our InfiniBand network, a job scheduler and an MPI implementation. We also chose a cluster management system to be able to change our cluster's setup quickly and to use a monitoring solution to be able to see issues and record performance. I present some of those software packages in the following subsections.

4.1 Operating System

Like most applications and drivers used in high performance computing, the required applications for the SCC were all thoroughly tested with the GNU Compiler Collection (GCC) and, due to Linux' ubiquity on clusters [16], had excellent Linux support. Linux systems are also heavily used at the the Steinbuch Centre for Computing, and we decided to use the same system used by the KIT. It was based on Suse Enterprise Linux Server 11 SP2 (SLES) and also includes their own cluster management called Kite. After some testing with the official OpenFabrics OFED stack included in SLES, I noticed its lack of support for FDR InfiniBand, so I switched to the Mellanox OFED stack, which is a modified version of the OpenFabrics OFED, to have FDR support.

4.2 Kite

Once the Operating system with the OFED stack was installed on our head-node, we configured Kite to be able to manage our cluster and reimage the nodes if needed. We configured our head-node as DHCP server and PXE boot server as a prerequisite for Kite and it also acted as a router for the compute-nodes' internal Ethernet network and as subnet-manager for the InfiniBand fabric.

All compute-nodes were configured to boot into PXE, which they either would leave immediately to boot from the local hard disk, or, in case a new OS image was available, would run Kite. Kite uses SystemImager [17] to create and install a new image. SystemImager creates a copy of the entire system it is installed on and creates a compressed tarball with Parallel BZIP2 (PBZIP2) [18]. On bootup, the PXE environment first copies and executes a small image that includes a Bittorrent client (aria2c), to download a small system that will execute all the operations needed to install the final system. Bittorrent is used to reduce the load on the head-node and avoid a bottleneck on its Ethernet connections by having each compute node distribute the data it already received to other nodes. Once the

small system is downloaded, it will format the drives and run any predefined scripts, for example setting the hardware clock, before mounting the drives. The actual system's image is also distributed with Bittorrent and is unpacked with PBZIP2 onto the local storage and scripts are executed to configure the nodes. After the system has been successfully installed, the node will reboot and start from the local drive.

Thanks to this system we could have our final cluster up and running in less then 20 minutes, as we had prepared our head-node a few weeks prior to getting the system. All it took was to plug in the Ethernet cables, add the compute-nodes' MAC addresses to the configuration, and start the imaging process.

4.3 FhGFS

For our distributed file system we used FhGFS or FraunhoferFS developed by the Fraunhofer Competence Center for High Performance Computing. It is a closed source but free to use file system. One of the advantages is "its distributed metadata architecture [that] has been designed to provide [...] scalability and flexibility" [19]. It can use InfiniBand to communicate or failover to Ethernet. We configured it to have data and metadata striped over all nodes excluding the head-node, that only acted as a client as its lower performance in InfiniBand throughput and memory bandwidth could have impacted the compute-nodes. We did not include any kind of redundancies, as we preferred simplicity over safety for our small system. As per recommendation of the FhGFS developers we formated the file system's data storage with XFS and the metadata storage with Ext4. We did not expect to gain much in performance due to the small size of our system, but nonetheless, we saw a 2-3% decrease in runtime for OpenFOAM compared to NFS.

4.4 Zabbix

To monitor the cluster, we used Zabbix [20], an open source monitoring solution. We monitored our Ethernet switch to see if there were any bottlenecks or other performance issues. Unfortunately our InfiniBand switch was unmanaged and we did not find another reliable way to check the performance and utilization of the InfiniBand network. With Zabbix we could provide live images of the current voltage and used power by polling our power measuring device and the competition's official power measuring PSU. We could also monitor temperature sensors, CPU load, RAM usage, file-system usage and much more. We configured Zabbix to have an agent running on each node that reported to a Zabbix proxy that was installed on our head-node. This proxy sent the information to the main Zabbix server running on one of our computers to plot graphs, make alerts, and provide other information like the layout of our network. Tests on one node running LINPACK with and without the activated agent showed no difference in performance.

4.5 SLURM

Once the cluster was running, we used SLURM [21] as resource manager. It is an open source project from LLNL, SchedMD, HP, and many others. Users submit jobs and indicate which resources they want. For example they can specify how many processes, nodes, how much RAM, or time is needed. SLURM allocates resources and schedules jobs once the requested resources are available. It also tracks the state of the cluster by monitoring how many nodes are allocated, are in a ready state, or are unavailable. In addition, SLURM can also do accounting on how many resources a user needed.

There are three ways to use SLURM for a user. They can use the `srun` command, which allocates the resources and executes the application with as many processes in parallel as specified. With `sbatch`/`salloc`, SLURM only allocates resources and then executes the specified program once, so the software needs to take care of spawning itself on the different

nodes. Srun and salloc block until the resources are freed and then write the output to the console, while sbatch returns immediately and once it is scheduled writes the application's output to files. In a typical cluster usage MPI is responsible for spawning the right processes on the correct nodes and users only schedule jobs to be executed and do not need immediate feedback written to their console, so in most cases sbatch is used.

SLURM supports scripts that can be run at different points during execution. I configured it to call echo 3 > /proc/sys/vm/drop_caches to clear file-system caches at the end of each job. This was because if the filesystem caches grew over about 90 GB, there was an issue with the Linux kernel not being completely NUMA aware. The caches were allocated in a non-uniform way over the two NUMA nodes which eventually lead to new memory allocations, for example by an application, on one node having to be moved to the other, thus creating a NUMA miss. This large number of NUMA misses lead to a significant drop in performance, which increased OpenFOAM's runtime from about 500 s to 1200 s. Clearing the caches after each job resolved the issue and should not have impacted the performance of the subsequent jobs, as they usually had different data. However, this was only a temporary fix, as a large enough job could fill up the file system caches and again cause the same problem.

4.6 Environment Modules

To manage the different installations of various versions of libraries, MPI, compilers, etc., we used environment modules, an open source project which "provides for the dynamic modification of a user's environment via modulefiles" [22]. For example, we installed various versions of GCC to their own directories and configured modules to change the $PATH environment variable to point to the version of GCC that was selected. It could also be used with libraries by changing the $LD_LIBRARY_PATH. We also configured Intel MPI with by setting its configuration environment variables in its modulefile. Thanks to modules we could easily and quickly switch compilers or MPI implementations and build the needed environment in scripts or our shell.

4.7 Intel MPI

For MPI we mainly used Intel MPI, as we expected a gain in performance from using its automatic tuning utility. Even though it is likely that most MPI implementations, if tuned correctly, have similar performance, we did not have the experience and time to do that properly. We chose to configure Intel MPI to have no fall-back option if the InfiniBand fabric failed, to avoid running into unexpected performance issues if InfiniBand was not available. As with our file system setup, we preferred failure of our cluster over a complicated setup with unexpected problems. We used the SHM:DAPL fabric to use shared memory for communication on the local node and InfiniBand RDMA for communication with other nodes. For MPI to spawn itself on other nodes we configured it to use SLURM to avoid having to distribute public keys on each node. As illustrated in Subsection 6.3, process pinning can lead to a significant performance increase so we enabled it explicitly.

The autotuner's default configuration runs over a multitude of benchmarks and tries every fabric; this process can potentially take days. We limited it to SHM:DAPL and it took around 1 hour to finish on our entire cluster. The autotuner can also be used to tune individual applications, however they should be configured to have a very short runtime as it takes too long otherwise. Unfortunately, time was too short to use the MPI tuning utility on our applications. The tuning job that I started on OpenFOAM was canceled after 14 hours with a job to be benchmarked that had a runtime of 10 minutes. I assume that a good test for the autotuner are 2 timesteps in OpenFOAM to ensure some kind of communication, maybe include a step to write to disk too. However reducing OpenFOAM's

grid resolution, to shorten time, is detrimental to the results as it does not simulate typical usage. The default settings created by the autotuner, that were not specific to OpenFOAM, had a negative impact on OpenFOAM, there was a 2-3% increase in runtime.

5 HPCC and LINPACK

The HPC Challenge Benchmark (HPCC) [23] is a benchmark used on clusters to test a wide variety of aspects that are important for a cluster's performance, like the raw processing power, but also the performance of the network and memory. It is a collection of benchmarks, to measure processing power it includes HPL, which is the LINPACK benchmark, DGEMM and FFT. LINPACK measures the floating point rate at solving a linear system of equations, DGEMM measures the floating point rate of execution of double precision real matrix-matrix multiplications, and FFT measures the floating point rate of execution with Discrete Fourier Transform. To benchmark the memory, HPCC includes STREAM to measure sustainable memory bandwidth and RandomAccess for the rate of integer random updates of memory. Finally it also uses PTRANS to test the total communications capacity of the network.

I configured HPCC according to an article published by Intel to use Intel MKL's BLAS and FFTW implementations [24]. HPCC can be configured to use larger problem sizes, by changing the "N" value of the configuration, to adapt it to the cluster's size. For most tests the only configuration is done via the "N" parameter, and most of HPCC's configuration file is for tuning HPL. Unlike most other benchmarks included in HPCC, HPL's performance is heavily influenced by the matrix size N. With $N = 1000$, we could only achieve 20 GFlops on one node, however, if it was increased to $N = 100000$, we could achieve over 200 GFlops. To configure HPL, we used a process distribution of $P = 13$ by $Q = 16$ due to our cluster's layout of 13 nodes with 16 cores each. At the competition there were issues with the power measurements that did not measure the current voltage, so we could not run the cluster at full capacity and we reached 2.27 TFlops with $N = 100000$ in HPCC and in a dedicated HPL run 2.37 TFlops with $N = 250000$. Once the size of N was reasonably large, tuning with N, the matrix size, and nb, the block size used in the algorithm, could gain a few GFlops, but the improvements were a few percent at best.

We had the following results with our cluster at the competition. We only had about 2.7 kW at our disposal because of the power measurement. We took 2nd place in LINPACK with 2.4 TFlops and were only beaten by a GPU cluster from NUDT which had 2.6 TFlops. We received the "best LINPACK on x86 cluster" award for this. Here are a few highlights of the HPCC run at the competition :

- 2.27 TFlops with HPL
- 109 GFlops with MPIFFT, the parallel MPI version of FFT
- SingleSTREAM add 7.27 GB/s memory bandwidth with 1 process
- StarSTREAM add 3.83 GB/s memory bandwidth with 16 independent processes per node
- 2.6 µs average ping pong latency with InfiniBand
- 5.3 GB/s ping pong bandwidth with InfiniBand

6 OpenFOAM

One of the required applications at the Cluster Competition was OpenFOAM [25], Open Source Field Operation and Manipulation, which I was tasked to optimize. It is an open

source set of libraries, distributed and promoted by the OpenFOAM foundation [26], that can be used to create Computational Fluid Dynamics (CFD) applications and comes with a wide variety of ready to use solvers and tutorial jobs. Solvers in OpenFOAM are programs that use the OpenFOAM libraries to calculate a certain kind of problem, like interFOAM for incompressible fluids, and can use linear solvers like GAMG or PCG to solve linear problems.

In the following subsections, I will first present my research into optimizing OpenFOAM to use GPUs, then in Subsection 6.2 I give a quick description of what compilers I used. In Subsection 6.3, I describe the effects that careful process distribution and pinning has on OpenFOAM's performance, followed, in Subsection 6.3, by some small observations I made while preparing OpenFOAM and conclude this section by comparing our results with our competitors in Subsection 6.4.

6.1 OpenFOAM on GPU

Early plans were to use a hybrid solution, so I investigated several ways to optimize OpenFOAM on the GPU. I found several implementations, an open source library called ofgpu [27] and two proprietary solutions, SpeedIT [28] and Culises [29]. All of them had the drawback that they did not support drop in replacements for all the linear solvers, which in our case was problematic, as we did not know which problems would be run at the competition. It was also unclear if we could use different linear algebra solvers than the ones defined in the system/fvSolution file provided during the competition. In the worst case we could not modify the problem to suit our needs, for example by replacing GAMG[1] with PCGGPU[2] as a linear solver and the AMG[3] preconditioner. At least SpeedIT needed to use the binaries provided by the vendor for the solvers, for example we would have to use their binary version of interFoam to be able to use their library. This could mean that in some cases we would have to run the problem on CPUs, as we either lacked binaries for the solver or linear algebra solver, leaving the GPUs idle. Moreover no data on speedup provided by the GPU solution providers was convincing enough to take such a risk. This was the main reason we decided against the usage of GPUs for OpenFOAM.

In retrospect I created a few profiles using Valgrind's Callgrind utility on some of the provided example applications, which counts the amounts of calls to a function and estimates the cycles needed to execute it. OpenFOAM uses a blockmesh to map the problem to cells that change their properties, this could be pressure, heat, velocity or anything the user wishes to simulate. It also divides the problem into discrete time-steps, for example you could have it calculate each millisecond and write the results to disk in steps of 10 ms. After each simulated time-step, the data is updated and then the processing of the next time-step begins, resulting in many iterations of the algorithm.

I profiled several tutorial jobs, icoFoam/cavity[4] with 400 cells, interFoam/laminar/damBreak with around 2300 cells, and solidDisplacementFoam/plateHole with 1000 or 100000 cells. I also profiled the larger applications of the competition, these were backStep with around 500000 cells, and square with 3 million cells, both using the icoFoam solver. Each was executed with just one process and on the latter two I only executed 1 time-step. This means that no communication step happened, which was still sufficient to determine if a GPU would be of benefit. I wanted to determine what kind of computations are performed and if according to those a GPU would be a viable choice. And in case that was true, I would have to profile the applications with more communications, like done by Michael

[1] Generalized geometric-algebraic multi-grid
[2] Preconditioned conjugate gradient on GPU
[3] Algebraic Multigrid
[4] The first directory in the path indicates the solver used in the tutorial applications

Moylesa, Peter Nash and Ivan Girotto [30] or Massimiliano Culpo [14]. Their studies determined that on our size of cluster around 30%-35% of the time would be spent on communications and 25%-35% on I/O, leaving in the best case only 45% of the time for computation, which would drastically reduce the percentages gathered while profiling just one time-step.

According to the cycle estimation of Valgrind, the cavity problem spent 3.5% running PBiCG::solve() and 10.4% running PCG::solve(), so about 14% of the time were the linear solvers. DamBreak spent 0,5% running PCG::solve(), the PBiCG solver ran too shortly to be measured. PlateHole with 1000 cells spent 13,2 % of its time in the GAMG solver, with the cell count increased to 100 thousand it was 20% of the time. BackStep used 50% of its time with GAMG and 1% with PBiCG, square 63% with GAMG, 1% with PBiCG.

From the data we can see on the plateHole test that the larger the dataset, the more time the application will spend with solving the linear algebra. It, however, also depends on what problem is solved, the smaller cavity problem spent a larger percentage of its time with the linear solvers than damBreak and plateHole in its default 1000 cell setup. In the case of backStep the maximum theoretical speedup, if we could accelerate the Linear algebra solver to be infinitely fast, would be two. However our tests showed that the power usage of one GPU was about the same as an entire node with only CPUs, so in this case there would be no gain in efficiency if we were running the linear algebra solver on a GPU. Tests done by FluiDyna [31] showed with a grid of 3M cells on one CPU, about 55% of the time was spent in the GAMG solver. Other linear solvers on the same problem made up a larger percentage of the execution time, however it is unclear on how those affected the overall runtime. Also the problems that are actually calculated by one core are much smaller than the total amount of cells, as the grid gets split up typically to about 30-50 thousand cells per core, which would likely reduce the share of time spent in the linear solver. Even ignoring communication and I/O, using the problems calculated at the competition, it is unlikely that we would have had a gain in efficiency with GPUs.

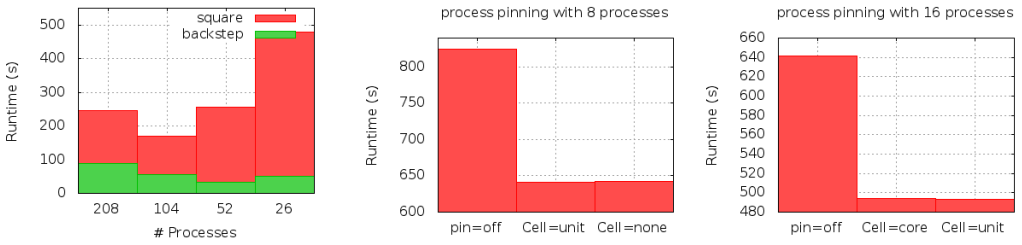
6.2 Compilers

Like many open source projects, OpenFOAM can easily be compiled with GCC (g++ for C++), as most testing is done with the free GNU compilers. I ran into a few issues with the Intel composer (icpc for C++), where versions after 12.1.0 would break the build. This is due to OpenFOAM's extensive use of the only recently released C++11 standard and most likely is actually based on a draft of C++0x, as no compiler has completely implemented the standard yet. GCC is quite popular, so it is likely that OpenFOAM uses GCC's incomplete implementation of C++11 for testing. Recent additions to Intel's compiler lead to a more complete implementation of C++11, and broke the build for OpenFOAM. It is often advisable to start out testing with GCC and then when you have more experience with the application test with other compilers.

Tests with both icpc and g++ showed no difference in execution speed, however as we had more information about tuning with icpc, I decided to use the Intel suite. It seems that the defaults of OpenFOAM were quite well tuned in the "Opt" build as any change on the compiler flags resulted in a drop of performance. For example using -xHOST instead of the default -xSSE3 to tune the binary to the machine resulted in an increase of 2% in runtime.

6.3 Process Distribution and Pinning

I did several tests on the KIT's IC1 cluster and on our cluster changing the distribution of processes. During the initial tests on the University's cluster, I kept the same amount of processes but used a different distribution over the cluster. With all 8 cores used per



(a) # of processes on the cluster

(b) Process pinning

Figure 6.1: Distribution of processes

node, the runtime for a modified damBreak problem with 64 processes took on average 1635 s. If I used only 4 cores per node, so only every second core, the runtime was 564 s on average. I nearly had a 3x speedup by doubling the number of nodes used, but only using half of the cores on each.

During the competition, I used fewer processes distributed over the same amount of nodes. As one can see in Figure 6.1a the runtime with the maximum amount of processes is not the best. I achieved the best runtime for backStep with 52 processes and around 10000 cells per process, with two cores per CPU used. With square the best runtime was achieved with 104 processes, four cores per CPU, with around 30000 cells per process. The runtime of backStep dropped from 88 seconds on 208 processes to 34 s with 52 processes, decreasing the runtime by about 60%. On square from 245 s with 208 to 169 s with 104 processes, decreasing runtime by about 30%.

As we can also see in HPCC's result for starSTREAM and singleSTREAM, the memory bandwidth is nearly doubled using only 1 core instead of having all 8 competing. Only in the latter case, where the amount of processes is changed, the increased amounts of I/O and communications are playing a role, as with the former the amount of processes and thus communication stayed stable. Intel's Turbo Boost, that increases the CPU's clock in certain cases, is not the deciding factor as it was disabled on the IC1 cluster. My theory is that while in the case of changing the amount of processes, the change in communications does influence runtime, that the main speedup in both cases is due to the higher memory bandwidth available for the single cores and therefore allows for a more effective processing. Like the work from Michael Moylesa, Peter Nash and Ivan Girotto [30] hints, these tests show that there is an optimal amount of cells per process, where the communications and I/O overhead, processing speed of the single threads and memory bandwidth are in an optimal balance. With the limited data I cannot make a prediction about how many cells lead to the most efficient execution time. It also seems to be dependent on the application that is run. I would have to do more tests to effectively reduce the search-space to find the optimal distribution. Right now a guess would be for our system, somewhere between 10000 and 30000 cells.

I also made tests with process pinning, to prevent processes to be scheduled on different processor cores during execution. The default of Intel MPI is to enable process pinning, while for OpenMPI is to disable it. The tests in Figure 6.1b were made with Intel MPI, but the results with OpenMPI were quite similar. The "cell" in Intel MPI defines what it should consider as independent processor cells to bind processes to, unit means any logical CPU, core means any physical core, and none means that we did not use that parameter specifically. The difference in runtime is quite noticeable with an improvement of about 25% for both 8 processes and 16 processes when process pinning was enabled. Also the standard deviation dropped between the runtimes from around 20 s with no pinning to

1 s with pinning. The cell parameter did not influence the runtime and if I looked at the distribution using htop, it was clear that in both cases all processes were mapped to physical cores and distributed evenly over the processors.

With the default Intel MPI setup, processes were wandering between the two logical processors on the physical core. The standard deviation between runtimes was higher and so was the average runtime. Defining anything as a cell mitigated that migration and stabilized the runtimes. Also if I used the defaults with anything more than 8 processes and less than 16, the processes were scheduled in a way to fill up one processor, in our case 8 processes for 8 cores, with the remaining processes on the other. I avoided this by using the map=scatter parameter to spread the processes out. Unfortunately, there was not enough time to thoroughly test the impact on the runtime these two modifications had, I theorize however that it was beneficial to avoid both scenarios. In case of just switching logical processor on the same core, caches are still valid, so there should be no or only a small performance penalty, however for each migration the scheduler has to interfere and a greater share of time is not used to process data. Moreover, an uneven spread of processes will likely impact the performance greatly as they have to be synchronized and wait for the slowest process to finish, and thus the processes on the congested CPU would slow everything down.

6.4 Other Notes

I gained a 2-3% speedup using FhGFS instead of NFS and by linking to a library provided by the FhGFS developers to buffer calls to stat(), I could gain a few seconds in runtime. These effects would probably be more noticeable on a larger system with more I/O.

I had a problem with the ptscotch decomposition method in the decomposeParDict, which is used by snappyhexmesh to map a 3d model to the blockmesh of a single process. If it is used with more than 128 processes, snappyhexmesh hung silently or, if I was lucky, crashed. So far I have found no solution. Neither changing the resolution nor using a multiLevel distribution, where one level for example has 4 domains that get split each into 64, solved the problem. However, at least now there is an error message and the jobs reliably crashes with a stacktrace, which should ease debugging.

I also want to note that the better our RAM, the closer the power usage of OpenFOAM approached that of HPL, so we could possibly gauge the efficiency of OpenFOAM by measuring the power usage. This however was just a small observation I made during tests and was not explored much further.

6.5 Results Compared to our Competitors.

A while after the competition we received the anonymized results. For OpenFOAM's square job we took 2nd place with our 33 s, and for backStep we took 3rd. Comparing our results to the team that was closest to us, I saw that we beat them with 33 seconds against their 51 s in square, but they beat our 169 s on backStep with 114 s. So we beat each other roughly by 35-40% runtime improvements.

I can only speculate on why that is. During the execution of our runs, our cluster was nowhere near the 3 kW we could use, the power usage was somewhere around 2.2 kW or less. This of course meant that we could have used the remaining 800 W with higher clocks, maybe more nodes too. I believe in this case however a higher CPU clock would have been more beneficial. Since only every second or even fourth core was used, each core had a much higher memory bandwidth. As I noticed, OpenFOAM is quite memory bound, which explains the only small runtime improvement in Table 1 with a higher CPU clock, a 45% increase in CPU clock lead to around 12% speedup, when all 8 cores are used.

However having twice the memory bandwidth would most likely have helped the cores perform closer to their maximum capacity, likely leading to a more significant speedup. I believe now that using CPUs with around 3 GHz would have benefited us here. The low usage of the CPU would have triggered its Turbo Boost, increasing the clock, and thus could use the memory increased bandwidth more effectively. I however did not test how the 2.6 GHz CPUs would behave compared the 1.8 GHz ones when they are only partially used, as we never intended to use only parts of the cluster's capacities.

7 Conclusion

Finally, I can say that my goal of presenting a working cluster with all of the known applications working was fulfilled. None of us had any prior experience with clusters, so I am even more astonished that we managed to do this in such a short time. Our final cluster was only ready 2 days before the competition due to bottlenecks in getting the parts. And despite all these issues, we won the "Best LINPACK on x86" and "Fan Favorite" award.

During the many tests, several effects manifested themselves. It seems apparent that high CPU clocks do not mean best efficiency, as shown by OpenFOAM running on the cluster and using every core. But if the workload is small enough that you can reduce the cores needed, higher clocks might be beneficial. The greatest gains in performance can be achieved by carefully choosing the best suited hardware for the application and effectively distributing the applications' processes over the system. Compiler optimizations might help if the application comes with a bad default, but often compiler optimizations risk falsifying the application's results. LINPACK for example does not compute with a sufficiently low error with some compiler options. I thus recommend first testing the cluster's hardware to match the applications and the goals.

Now that we have a better understanding of the goals, which is to run small jobs as fast as possible, for a future project like this I would most likely choose CPUs with a higher clock, maybe fewer nodes. If we exceed the power limit, it is easy to clock CPUs down, this is considering the 10% higher power usage on low clocks compared to actual low power CPUs. Most likely only the LINPACK score would suffer from the decreased efficiency, an acceptable trade-off to speed up applications.

Bibliography

[1] TOP500 Introduction and Objectives. http://www.top500.org/project/introduction.

[2] ISC'12 Student Cluster Competition. http://www.hpcadvisorycouncil.com/events/2012/ISC12-Student-Cluster-Competition/.

[3] Student Cluster Competition Rules. http://www.hpcadvisorycouncil.com/events/2012/ISC12-Student-Cluster-Competition/rules.php.

[4] Student Cluster Competition FAQ. http://www.hpcadvisorycouncil.com/events/2012/ISC12-Student-Cluster-Competition/attendee_reg.php.

[5] Einar Rustad. NumaConnect. A high level technical overview of the NumaConnect technology and products. http://www.numascale.com/numa_pdfs/numaconnect-white-paper.pdf.

[6] Carlos Sosa and Brant Knudson. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM, fourth edition, August 2009.

[7] Bob Walkup. *Blue Gene/P System and Optimization Tips*. IBM Watson Research Center.

[8] TOP500 List June 2011. http://www.top500.org/list/2011/06/100.

[9] AMD Opteron 6100 models. http://products.amd.com/en-us/OpteronCPUResult.aspx?f1=AMD+Opteron%E2%84%A2+6100+Series+Processor.

[10] HPC Advisory Council. OpenFOAM Performance Benchmark and Profiling. http://www.hpcadvisorycouncil.com/pdf/OpenFOAM_Analysis_and_Profiling_Intel.pdf, October 2010.

[11] HPC Advisory Council. Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing. http://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf, 2009.

[12] Mellanox Technologies. *216 Port InfiniBand FDR Switch Platform Hardware User Manual*, 2012.

[13] *Intel Server Board S2600CP Family Technical Product Specification*.

[14] Massimiliano Culpo. Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters. Technical report, Partnership for Advanced Computing in Europe, 2011.

[15] Supermicro. *X9DR3-F, X9DRi-F USER'S MANUAL*.

[16] Operating System share for 06/2011. http://top500.org/charts/list/37/os.

[17] SystemImager. http://systemimager.sourceforge.net/.

[18] Parallel BZIP2. http://compression.ca/pbzip2/.

[19] FraunhoferFS. http://www.fhgfs.com/cms/.

[20] Zabbix. http://www.zabbix.com/.

[21] SLURM: A Highly Scalable Resource Manager. http://www.schedmd.com/slurmdocs/.

[22] Environment Modules Project. http://modules.sourceforge.net/.

[23] HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/.

[24] Vipin Kumar. Use of Intel MKL in HPCC benchmark. http://software.intel.com/en-us/articles/performance-tools-for-software-developers-use-of-intel-mkl-in-hpcc-benchmark/, May 2012.

[25] OpenFOAM. http://www.openfoam.com/.

[26] The OpenFOAM Foundation. http://www.openfoam.org/.

[27] GPU v0.2 Linear Solver Library for OpenFOAM. http://www.symscape.com/gpu-0-2-openfoam.

[28] SpeedIT. http://speedit.vratis.com/.

[29] Culises. http://www.fluidyna.com/content/culises.

[30] Michael Moylesa, Peter Nash, and Ivan Girotto. Performance Analysis of Fluid-Structure Interactions using OpenFOAM. Technical report, Partnership for Advanced Computing in Europe, 2012.

[31] FluiDyna. Revolutionising high performance computing: Employing Fluidyna's hard- and software solutions based on Nvidia Tesla, 2012.